TP 1

OPEN-MP.

Exo 1:

chaque thread
1) → imprime son identifiant, afficher "Hello World" par un seul thread,
2) varier le nb de threads : OMP_NUM_THREADS, fonction omp_set_num_threads(), clause num_threads()
+ ordre de préséance

↳
```
#include <cstdio>
#include <iostream>
#include "omp.h"

int main() {
    omp_set_num_threads(5)   ↘ écrasé
    #pragma omp parallel num_threads(7) {
        printf("Thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
    #pragma omp sections {
        #pragma omp section {
            printf("Hello depuis thread %d\n", omp_get_thread_num());
        }}}
    return 0;}
```

ordre de préséance :
1) num_threads(n) (sur le #pragma omp parallel)
2) omp_set_num_threads(n)
3) OMP_NUM_THREADS

Exo 2:

a) initialiser A[n] tq A[i] = i pour $0 \leq i < N$
b) calculer la somme des elts de A[n]
c) paralléliser avec #pragma omp for
d) // avec #pragma omp sections: 4 sections, N/4 itérations
e) Comparer le temps d'exécution séquentielle et le temps d'exécution parallèle du programme

```
#include <chrono>
#include <iostream>
#include <vector>
#include "omp.h"

int main() {
    int N = 1000 000;
    std::vector<double> A(N);
    double somme = 0.0;
    auto start = std::chrono::high_resolution_clock::now();

#pragma omp parallel num_threads(4)
    { #pragma omp for
        for (int i=0; i<N; i++) {
            A[i] = (double) i;
        }
```

```
// somme avec omp for
#pragma omp for reduction (+:somme)
  for (i=0; i<N; i++) {
      somme += A[i]
  }
//    somme avec sections
#pragma omp sections {
    #pragma omp section {
        double sommeLocale = 0;
        for(int i=0;i<N/4;i++) { sommeLocale += A[i]; }
      #pragma omp atomic
        somme += sommeLocale;
     }
    #pragma omp section {
        double sommeLocale = 0;
        for( int i= N/4 ,i<(2*N)/4;i++) { sommeLocale += A[i]; }
     #pragma omp atomic
        somme += sommeLocale;
     }
    #pragma omp section {
        double sommeLocale = 0;
        for( int i=(2*N)/4 ;i<(3*N)/4 ;i++) { sommeLocale += A[i]; }
    #pragma omp atomic
        somme += sommeLocale;
     }
    #pragma omp section {
        double sommeLocale = 0;
        for(int i=(3*N)/4 ;i<N ;i++) { sommeLocale += A[i]; }
    #pragma omp atomic
        somme += sommeLocale;
     }
  }
} // fin omp parallel
std::cout << "La somme est " << somme << std::endl;
std::chrono:: duration<double> temps = std::chrono::high_resolution_clock::now() - start;
std::count << "Temps de calcul : " << temps.count() << "s\n";
return 0;
}
```

Exo3: _____

merge sort

- créer une région parallèle avec 4 sections qui trie chacune N/4 elts consécutif de A[N] avec std::sort
- + 2 sections chacune fusionnant deux tableaux triés de taille N/4 en un seul tableau trié de taille N/2 sur temp[N] (merge: code fourni)
- fusionner les 2 tableaux de taille N/2 en A[N] trié en dehors de la région parallèle.

```
# include <cstdio>
# include <iostream>
# include <cstdlib>
# include <algorithm>
# include <vector>
# include <chrono>
# include "omp.h"

int main() {
    std::srand(123);
    int N= 1000 000;
    std::vector<int> A(N);
    std::vector <int> temp(N);

    for(int i=0; i<N; i++){
        A[i] = std::rand();
    }
    auto start = std::chrono::high_resolution_clock::now();
    # pragma omp parallel num_threads(4) {
        # pragma omp sections {
            # pragma omp section {
                std::sort (A.begin(), A.begin()+ N/4);
            }
            # pragma omp section {
                std::sort (A.begin()+N/4, A.begin+ N/2),
            }
            # pragma omp section {
                std::sort (A.begin()+N/2, A.begin()+3*N/4);
            }
            # pragma omp section {
                std::sort (A.begin()+3*N/4, A.begin()+N);
            }
    } // fin omp sections

    # pragma omp sections {
        # pragma omp section {
            merge (&temp[0], &A[0], N/4, &A[N/4], N/2-N/4);
        }
        # pragma omp section {
            merge (&temp[N/2], &A[N/2], 3*N/4-N/2, &A[3*N/4], N-3*N/4);
        }
    } // fin omp sections
}
merge( &A[0], &temp[0], N/2, &temp[N/2], N-N/2);
std::chrono::duration<double> temps = std::chrono::high_resolution_clock::now() -start;
std::cout<< "temps de calcul:" << temps.count() << " s\n";
assertExo(A);
return 0;
}
```

```
void merge (int *res
            const int *tabA,
            const int sizeA,
            const int *tabB,
            const int sizeB)
```

Exo 4:

$$\pi \approx \int_0^1 \frac{4}{1+x^2}\,dx \approx \sum_{i=0}^{N-1} s \times \frac{f(i \times s) + f((i+1) \times s)}{2}$$

- code séquentiel
- parallélisation avec #pragma omp for // + test de performance
- chaque thread : N/P pour calculer piLocal => les additionner dans pi (variable partagée)

```
#include <chrono>
#include <iostream>
#include "omp.h"

inline double f(double x) { return (4/(1+x*x)); }
int main() {
    const int N = 100 000 000
    double pi = 0.0;
    double s = 1.0/(double)N

    // • Calculer le pi en séquentiel
    auto start = std:: chrono:: high_ resolution_clock ::now();
    for (int i=0; i<N; i++) {
        pi = pi + s*( f(i*s) + f((i+s)*s))/2 ; }
    std ::cout<< "pi = " << pi << std::endl;
    std:: chrono:: duration <double> tempsSeq= std :: chrono:: high_ resolution _clock ::now( ) - start ;
    Std :: cout << " Temps séquentiel : "<< tempsSeq . count()<< "s\n";
    // reset
    pi = 0.0; start= std:: chrono : : high_ resolution _clock :: now() ;
    // Calculer le pi avec omp for et reduction
    # pragma omp parallel {
        #pragma for reduction(+:pi) {
            for(int i =0; i<N; i++){
                pi = pi+ s*( f(i*s) + f((i+s)*s))/2 ; }
    } }
    std ::cout<< "pi = " << pi << std::endl;
    std.: chrono:: duration <double> tOMPfor = std :: chrono:: high_ resolution _clock ::now( ) - start ;
    Std :: cout << " Temps OMP for : "<< + OMPfor . count()<< "s\n";
    (reset..)
    // Calculer le pi avec la boucle faite à la main
    # pragma omp parallel {
        int thId = omp_get_thread_num();
        int numTh = omp_ get_num_threads();
        int elemParTh = N/numTh;
        int thDebut = thId * elemParTh;
        int thFin;
        // si N n'est pas divisible par numTh, le dernier thread parcour jusqu'au bout
        if (thid == numTh-1) {thFin = N;}
        else {thFin = thDebut + elemParTh; }
        double piLocal = 0.0;
        for(int i = thDebut; i<thFin ; i++) { piLocal = piLocal + s *( f(i*s) + f((i+1)* s)) /2; }
        #pragma omp atomic
            pi += piLocal; } //celle de atomic
    std:: cout << ... ;
    return 0; }
```

TP2 MPI

Exo1:

Affiche "Hello World", le rang et le nombre total de processus

```
#include <cstdio>
#include "mpi.h"

int main(int argc,char**argv){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world du processus %d/%d \n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Exo2:

- processus rang pair: envoyer un message contenant le rang du processus courant au processus impair correspondant. Recevoir le message du processus impair et l'afficher.
- processus rang impair: recevoir le message du processus pair associé, puis envoyer un message contenant la valeur reçue + 10 x le processus courant.

```
#include<cstdio>
#include <iostream>
#include "mpi.h"
using namespace std;

int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int msg;
    if((rank%2==0) && (rank+1<size)) { // le processus est pair et son binôme existe
        msg=rank;
        printf("Processus %d message envoyé au processus %d: %d\n", rank,rank+1, msg);
        MPI_Send(&msg,1,MPI_INT, rank+1, 0, MPI_COMM_WORLD);
        MPI_Recv(&msg,1,MPI_INT, rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Processus %d message reçu du processus %d: %d\n", rank, rank+1, msg);
    } else if(rank%2==1) { // le processus est impair
        MPI_Recv(&msg,1,MPI_INT, rank-1,0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        msg+=10*rank;
        MPI_Send(&msg,1,MPI_INT, rank-1, 0, MPI_COMM_WORLD);
    } else { //le processus est pair mais son binôme n'existe pas, donc il ne fait rien
    }
MPI_Finalize();
return 0;}
```

Exo 3:

(Calcul de pi, 4e formule)

P processus tq chaque processus effectue le calcul concernant N/p valeurs consécutives de i, puis fusionne son résultat avec ceux des autres. Au final, tous les processus doivent avoir la valeur de π.

```
#include <iostream>
#include "mpi.h"
inline double f(double x) { return (4/(1+x*x)); }

int main( int argc, char **argv){
    MPI_Init(&argc, &argv);
    int i;
    const int N = 100 000 000;
    double pi = 0.0;
    double s = 1.0/(double)N;

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calculer le pi tq chaque processus parcours N/size iteriations de la boucle. A savoir:
    // P0: (0 ___ N/size-1)
    // Pi: (N/size ___ 2N/size-1) etc.

    int begin=(N/size)*rank;
    int end = (rank == size-1) ? (N/size)*(rank+1);
    double piLocal=0.0;
    for(int i=begin; i<end; i++){
        piLocal += s*(f(i*s)+f((i+1)*s))/2;
    }
    if (rank==0) {
        pi = piLocal;
        for(int i=1;i<size;i++){
        MPI_Recv(&piLocal,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        pi += piLocal; }
        for(int i=1; i<size; i++) {
            MPI_Send(&pi,1,MPI_DOUBLE, i,0,MPI_COMM_WORLD);
        }
    } else{
        MPI_Send(&piLocal, 1,MPI_DOUBLE, 0,0, MPI_COMM_WORLD);
        MPI_Recv(&pi,1,MPI_DOUBLE, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank: %d, pi: %lf\n", rank, pi);
    }
    if(rank==0) {printf("Pi: %lf\n", pi);}
    MPI_Finalize();
    return 0;
```

Exo4:

- fusionne tableaux 2 par 2 ds les te P/2 processus, P/4, ——, 0 ↖ contiendra le tableau final trié A de taille NP (nb de processus tjs puissance de 2)

code de la fusion uniquement:

```
int count = N;
for(int r = size, r>1, r=r/2) {
  if (rank<r/2) {
    int partner = rank + r/2;
    std::vector<double> tempRecu(count);
    std::vector<double> tempsMerge(count+count);
    MPI_Recv(&tempsRecu[0], count, MPI_DOUBLE, partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    count *=2;
    std::swap(A, tempMerge); // le tableau fusionné devient mon tableau pour l'étape suivante
  }else if (rank>=r/2 && rank<r) { //j'envoie mon tableau à mon partenaire
   int partner = rank - r/2;
   MPI_Send(&A[0], count, MPI_DOUBLE, partner, 0, MPI_COMM_WORLD);
   break;
} }
```

TP 3 Avx

Exo1:

- Allouer A et B float[N] puis initialiser A[i] = i
- fct copie le contenue de A dans B (version non vectorisé + vectorisé)

```
// Allouer et initialiser deux tableaux de flottants de taille dim
float *tab0 = (float *) malloc(dim * sizeof(float));
float* tab1 = (float*) malloc (dim * sizeof (float));

for(int i=0; i<dim; i++ {
    tab0[i] = i;
    tab1[i] = 0, }

// Copier tab0 ds tab1 de manière scalaire (code non-vectorisé)
for (int i=0; i<dim; i++){ tab1[i] = tab0[i]; }

// copier tab0 dans tab1 de manière vectorisée avec AUX
int i;
for(i=0; i< dim -7; i+=8) {
   __m256 r0 = _mm256_loadu_ps(tab0+i);
   _mm256_storeu_ps(tab1+i, r0);
 }
  // copier le dernier bout sans vectorisation
 for(; i<dim;i++) {tab1[i] = tab0[i]; }

// Copier tab0 dans tab1 de manière vectorisée avec AUX et déroulement de facteur 4
int i;
for (i=0; i< dim -31 ; i+= 32){
    __m256 r0 = _mm256_loadu_ps(tab0+i);
    __m256 r1 = _mm256_loadu_ps(tab0+i+8);
    __m256 r2 = _mm256_loadu_ps(tab0+i +16);
    __m256 r3 = _mm256_loadu_ps(tab0+i +24);

    _mm256_storeu_ps(tab1+i, r0);
   _mm256_storeu_ps(tab1+i+8 , r1 ) ;
   _mm256_storeu_ps(tab1+i +16 , r2 ) ;
   _mm256_storeu_ps(tab1+i +24 , r3 );
}
// copier le dernier bout sans vectorisation
for(; i<dim;i++) {tab1[i] = tab0[i]; }

// désallouer
free(tab0);
free(tab1);
return 0; }
```

Exo 2 :

but : calculer $x^T y = \sum_{i=1}^{N} x_i y_i$

- allouer x, y tabs de float de taille N (divisible par 8) + initialisation (voir temps d'exec / performances)
- produit scalaire : - fct non vectorisée
  - fct vectorisée
  - vectorisée qui utilise fused-multiply-add (FMA)
  - déroulement de la boucle par un facteur de 2 et 4 (i.e. effectue 2 ou 4 itérations de la version précédente dans une seule itération)
  - (+ ftree-vectorise)

version séquentielle :

```
inline float produitScalaire( float *A, float *B, int taille ) {
    float res = 0.0f
    for(size_t i=0 ; i< taille; i++) {
        res += A[i] * B[i];
    }
    return res;
}
```

Version AUX :

```
inline float produitScalaireAUX (float *A, float *B, int taille) {
    float zero[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    __m256 res = _mm256_loadu_ps(zero);
    for(size_t i = 0;  i <taille; i+=8) {
    __m256 la = __m256_load_ps (A+i);
    __m256 lb = __m256_load_ps (B+i);
    __m256 lab = __m256_mul_ps (la, lb);
    res = _mm256_add_ps (res, lab);
}
float resTab[8];
__m256_storeu_ps(resTab, res);
return resTab[0]+ resTab[1] + resTab[2]+ resTab[3] +
       resTab[4] + ... + resTab[7];
}
```

(*)

version AVX+ FMA :

```
inline float produitScalaire AUXFMA(float *A, float *B, int taille){
    float zero[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    __m256 res = _mm256_loadu_ps(zero);
    for(size_t i = 0;  i <taille; i+=8) {
    __m256 la = __m256_load_ps (A+i);
    __m256 lb = __m256_load_ps (B+i);
    res = _mm256_fmadd_ps (la,lb, res);
}(*)
```

version AVX+ FMA + déroulement (uncrolling)

```
inline float produitScalaireAVXFMADEROULEMENT(float *__restrict__A, float *__restrict__B, int taille){
    float zero[8] = {0,0,0,0,0,0,0,0}
    __m256 res1 = _mm256_loadu_ps(zero);
    __m256 res2 = _mm256_loadu_ps(zero);
    __m256 res3 = _mm256_loadu_ps(zero);
    __m256 res4 = _mm256_loadu_ps(zero);

    for(size_t i=0; i<taille; i+=32) {
    __m256 pa1 = _mm256_loadu_ps(A+i);
    __m256 pa2 = _mm256_loadu_ps(A+i+8);
    __m256 pa3 = _mm256_loadu_ps(A+i+16);
    __m256 pa4 = _mm256_loadu_ps(A+i+24);
                                    ){
    __m256 pb1 = _mm256_loadu_ps(B+i);
    (de même pb2, pb3, pb4)
    res1 = _mm256_fmadd_ps(pa1, pb1, res);
    res2 = _mm256_fmadd_ps(pa2, pb2, res);
    res3 = _mm256_fmadd_ps(pa3, pb3, res);
    res4 = _mm256_fmadd_ps(pa4, pb4, res);
}
res1 = _mm256_add_ps(res1, res2);
res3 = _mm256_add_ps(res3, res4);
res1 = _mm256_add_ps(res1, res3);
(*) }
```